Homework 4: Meal Ordering System

For this assignment, you will implement a Meal Ordering System that allows customers to order meals from different restaurants. Additionally, you will write and fix test cases to ensure that every step in the checkout process—placing the order and having the order processed—works correctly.

Important: Review the starter code thoroughly before beginning this assignment. Understanding how the classes interact with each other on a high level is crucial, especially within the object-oriented programming paradigm. Feel free to draw a diagram or jot down notes on how each class and method connects.

Overview

Customer Class

Represents a customer who wants to order meals from a restaurant. You will implement one key method in this class: `place_order()`.

Instance Variables

- `name (str)`**: The customer's name.**
- `employer_id (int or None)`**: ID of the restaurant the customer works for. If not employed by a restaurant, this is** `None`**.**
- `account_balance (float)`**: Amount of money in the customer's account (default is $15.0).**

Provided Methods

- `__init__(self, name, employer_id=None, account_balance=15.0)`**: Initializes the customer's attributes.**
- `__str__(self)`**: Returns the customer's name and current account balance as a string.**
- `deposit_funds(self, amount)`**: Adds the specified amount to the customer's** `account_balance`**.**

To Implement

- `place_order(self, restaurant, order_dict)`

- **Parameters:**
  - `restaurant (Restaurant object)`: The restaurant to order from.
  - `order_dict (dict)`: A dictionary where:
    - Keys: `MenuItem` objects.
    - Values: Dictionaries containing:
      - `quantity (int)`: Number of that `MenuItem` requested.
      - `express (bool)`: Whether the order should be an express order.
- **Process:**
  - Calculate Total Cost:
    - Use the restaurant's `calculate_item_cost()` method to compute the total cost of each item in the order.
    - Important: Pass `customer=None` to avoid applying employee discounts during order placement.
    - Sum up the total cost of all items.
  - Check Funds:
    - Verify if the customer has enough funds in `account_balance` to cover the total cost.
    - If not, immediately return `False`.
  - Process Order:
    - Call `restaurant.process_order(order_dict)`.
    - If `process_order` returns `True`:
      - Deduct the total cost from the customer's `account_balance`.
      - Call `restaurant.accept_payment(total_cost)` to add the amount to the restaurant's income.
      - Return `True`.
    - Otherwise, return `False`.

MenuItem Class

Represents a single meal or item on the restaurant's menu.

Instance Variables

- `name (str)`: The meal's name (e.g., "Cheeseburger").

Provided Methods

- `__init__(self, name)`: Initializes the `MenuItem` object's name.

- `__str__(self)`: Returns the name of the menu item as a string.

To Implement

- `__eq__(self, other)`
  - Purpose: Check equality based on the item's name.
  - Parameters:
    - `other (MenuItem)`: Another `MenuItem` instance to compare.
  - Behavior:
    - Return `True` if the `name` attributes of both `MenuItem` instances are the same.
    - Otherwise, return `False`.
- `__hash__(self)`
  - Purpose: Return the hash based on the item's name.
  - Behavior:
    - Return a hash of the `name` attribute to allow `MenuItem` instances to be used as dictionary keys.

Restaurant Class

Represents a restaurant that can fulfill orders.

Instance Variables

- `name (str)`: The restaurant's name (e.g., "Pizza Palace").
- `restaurant_id (int)`: Unique ID of the restaurant.
- `income (float)`: Amount the restaurant has earned so far (default is $0.0).
- `menu (dict)`: Keys are `MenuItem` objects; values are quantities (inventory) available of each `MenuItem`.
- `prices (dict)`: Keys are `MenuItem` objects; values are the prices specific to this restaurant for each `MenuItem`.

Provided Methods

- `__init__(self, name, restaurant_id, income=0.0)`: Initializes the restaurant's attributes, including empty `menu` and `prices` dictionaries.
- `__str__(self)`: Returns the restaurant's name and current total income.
- `accept_payment(self, amount)`: Adds `amount` to the restaurant's `income`.
- `set_price(self, item, price)`: Sets the price for a specific `MenuItem`.
- `get_price(self, item)`: Retrieves the price for a specific `MenuItem`.

To Implement

- `calculate_item_cost(self, item, quantity, express, customer=None)`
  - **Purpose: Returns the total cost for a given `MenuItem`, considering quantity, express ordering, and optional employee discounts.**
  - **Parameters:**
    - `item (MenuItem)`**: The item being ordered.**
    - `quantity (int)`**: Number of the item being ordered.**
    - `express (bool)`**: Indicates whether this is an express order.**
    - `customer (Customer or None)`**: The customer placing the order; used for checking employee discounts.**
  - **Behavior:**
    - **Base Cost: Retrieve the item's price from `self.prices[item]` and multiply by `quantity`.**
    - **Express Surcharge: If `express` is `True`, multiply the base cost by 1.2 (adding a 20% surcharge).**
    - **Employee Discount:**
      - **If `customer` is provided and `customer.employer_id` equals `self.restaurant_id`, apply a 40% discount to the new total (after any express surcharge).**
    - **Return: The computed total cost as a float.**
- `stock_up(self, item, quantity)`
  - **Purpose: Increases the restaurant's menu inventory for a specific item.**
  - **Parameters:**
    - `item (MenuItem)`**: The item to add.**
    - `quantity (int)`**: Number to add to the existing stock.**
  - **Behavior:**
    - **If `item` already exists in `self.menu`, add `quantity` to the existing amount.**
    - **Otherwise, create a new entry in `self.menu` with `item` as the key and `quantity` as the value.**
- `process_order(self, order_dict)`
  - **Purpose: Checks if the restaurant can fulfill the entire order and, if so, reduces the appropriate inventory.**
  - **Parameters:**
    - `order_dict (dict)`**: Dictionary where keys are `MenuItem` objects and values are dictionaries of the form `{'quantity': X, 'express': Y}`.**
  - **Behavior:**
    - **Check that every requested item and quantity is available in `self.menu`.**

- If any item is out of stock or not present in `self.menu`, return `False` immediately (no partial fulfillments).
- If the order can be fulfilled in full, subtract the requested quantities from the restaurant's menu and return `True`.

Example Order Dictionary

```
# Example of an order_dict that might be passed:
order_dict = {
    MenuItem("Cheeseburger"): {"quantity": 2, "express": False},
    MenuItem("Fries"):        {"quantity": 3, "express": True}
}

# The outer dictionary's keys are MenuItem objects.
# Each key points to another dictionary specifying "quantity" and "express" status.
```

Note: Since each `Restaurant` maintains its own pricing for `MenuItem` objects, ensure that prices are set appropriately for each restaurant using the `set_price()` method before processing orders.

Tasks to Complete

1. Customer Class (15 Points)

Implement the `place_order(self, restaurant, order_dict)` method.

- **Process:**
  - **Calculate Total Cost:**
    - Use `restaurant.calculate_item_cost(item, details['quantity'], details['express'], customer=None)` for each item in `order_dict`.
    - Important: Pass `customer=None` to avoid applying employee discounts during order placement.
    - Sum up the total cost of all items.
  - **Check Funds:**
    - If `self.account_balance < total_cost`, return `False`.
  - **Process Order:**

        - Call `restaurant.process_order(order_dict)`.
        - If it returns `True`:
            - Deduct `total_cost` from `self.account_balance`.
            - Call `restaurant.accept_payment(total_cost)` to add the amount to the restaurant's income.
            - Return `True`.
        - Otherwise, return `False`.

2. MenuItem Class (5 Points)

Implement the following methods:

- `__eq__(self, other)` **(4 Points)**
    - **Purpose: Check equality based on the item's name.**
    - **Parameters:**
        - `other`**: Another** `MenuItem` **instance to compare.**
    - **Behavior:**
        - **Return** `True` **if the** `name` **attributes of both** `MenuItem` **instances are the same; otherwise, return** `False`**.**
- `__hash__(self)` **(1 Point)**
    - **Purpose: Return the hash based on the item's name.**
    - **Behavior:**
        - **Return a hash of the** `name` **attribute to allow** `MenuItem` **instances to be used as dictionary keys.**

3. Restaurant Class (25 Points)

Implement the following methods:

- `calculate_item_cost(self, item, quantity, express, customer=None)` **(10 Points)**
    - **Implement logic for:**
        - **Base cost, express surcharge (20%), and optional employee discount (40%).**
- `stock_up(self, item, quantity)` **(5 Points)**
    - **Add to or create: A new entry in the restaurant's menu.**
- `process_order(self, order_dict)` **(10 Points)**
    - **Verify full stock availability: Then reduce stock or return** `False`**.**

4. Write and Fix Test Cases (15 Points)

- **Write Test Cases for the Following Scenarios in** `test_place_order` **(12 Points):**

    - The customer doesn’t have enough money in their account to place the order.
        - Expected Outcome: `place_order` should return `False`.
    - The restaurant doesn’t have enough inventory for an item.
        - Expected Outcome: `place_order` should return `False`.
    - The restaurant doesn’t carry an item mentioned in the order.
        - Expected Outcome: `place_order` should return `False`.
- Fix the Test Cases in `test_place_order_2` (3 Points):
    - Important: Ensure that after a successful order:
        - The customer's account balance is reduced by the correct amount.
        - The restaurant's income increases by the correct amount.

Grading Rubric (60 Points Total)

- Customer Class (15 Points):
    - `place_order` method correctly implemented: 15 points
- MenuItem Class (5 Points):
    - `__eq__` method correctly implemented: 4 points
    - `__hash__` method correctly implemented: 1 point
- Restaurant Class (25 Points):
    - `calculate_item_cost` correctly implemented: 10 points
    - `stock_up` correctly implemented: 5 points
    - `process_order` correctly implemented: 10 points
- Test Cases (15 Points):
    - `test_place_order`: 3 new scenarios (4 points each): 12 points
    - Fixing `test_place_order_2`: 3 points

Extra Credit (6 Points)

- Extend `calculate_item_cost` to apply an extra discount rule:
    - If `customer.employer_id` matches `self.restaurant_id`, the customer gets a 40% discount after any express surcharge.
    - If `express` is `True`, the 20% upcharge applies first, then the 40% discount is applied on the new total.
- Implement this logic carefully to earn up to 6 additional points.

Final Notes

- **Avoid Hardcoding: Do not hardcode expected values (like forcing tests to pass by setting attributes directly).**
- **Modify Tests Carefully: Only modify tests where explicitly allowed.**
- **Testing: If you have partial progress on one test, you may comment out others while you debug, but ensure all tests are uncommented before final submission.**

Good luck and happy coding!
